TerraWeather
Ensuring Your Safety & Success

AI Workshop – Lecture 7

Deep Learning & Stacked Autoencoders

7.0 Classic Networks

When perceptrons were first invented, it was quickly realized that while a single layer of perceptrons could model only some classification tasks, just 2 layers would allow them to become universal classifiers (ie, train perfectly on any classification task).

This led to the “classic” network configuration:

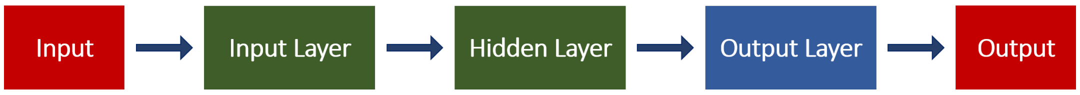

Figure 7.0 – “Classic” NN architecture. The first two (green) boxes represent perceptron layers (combiner + saturator), while the output layer (blue) is a combiner only.

Without the *hidden layer* in Figure 7.0, the network can only model convex classification tasks, (ie, tasks where the classification boundary is a convex function).

“Deeper” networks – with more hidden layers – were not much explored because:

1. **No Value:** They were thought to add no value, since 2 layers were sufficient for a universal classifier / approximator,

2. **Hard to Train:** Deeper layers are hard to train correctly due to the vanishing/exploding error problem of the BP algorithm,

3. **Slow to Train:** Deeper networks are slower to train compared to shallower networks of the same size. Remember that in the early 1990's even supercomputers only had several GB of RAM, and very slow speeds compared to the present.

These problems caused practitioners to largely abandon deep layers in favor of wide shallow layers. Much of the literature in applied NNs from the 1990's and early 2000's all use variations on the classic 2-layer configuration.

(2) was especially problematic for recurrent networks (those containing a cyclic path – see Lecture 3), which are trained using a special technique called *backpropagation in time*, which required mapping a recurrent network into an deep network with shared weights, the depth of the network representing time.

Quiz 7A:

1. * Explain clearly why deeper networks are slower to train?
2. Explain why recurrent networks are prone to vanishing/exploding errors during BP learning. Why should this phenomenon be a problem for learning?

7.1 The Value of Deep Networks

Neuroscientists have long known that real brains in the animal world are composed of “deep” layers of 100s of neurons.

In the 1980s – 1990s, it was also found that neurons fire in clusters, in response to certain *features* in sensory input. These clusters are named *feature detectors* , and are the way the neurocortex (the part of the brain that processes sensory information) breaks down sensory input – sight, sound, smell, tough and taste – to be processed.

These two key facts point to the value of networks many layers deep – they allow the formation of feature detectors that could greatly improve generalization.

To understand why this is so, if you want to recognize a face, you'd want to break this down into separate *features* – a nose, eye, hair, mouth, etc. These features in turn can be broken into further sub-features – a mouth has lips, teeth, etc.

This *hierarchy of features* is obviously commonplace. The neurons that detect these features also roughly correspond to this hierarchy. Neurons in bottom layers closer to the input perform basic signal processing (eg, edge detection for vision) while those in higher layers combine these signals to detect sub-features and even higher layers combine these sub-features into complex combinations like a face.

This reasoning clearly shows that deep layers are capable of much better generalization, much more so than shallow networks of comparable size.

7.2 Training Deep Networks

While this potential of deep networks was well-known even in the 1990s, the question was how to train them.

Nature does not use backpropagation to “train” neurons! So, the deep training problem with BP meant that no great progress was made on the subject for a long time.

About 10 years ago, a breakthrough was made that eventually sparked the “Deep Learning” revolution. In the years that followed, this was distilled into two ideas:

1. The value of deep layers is tied intimately to feature detectors. So, we need to train a special type of deep network called an *autoencoder,* which we'll describe shortly.

2. You can train deep autoencoders layer-by-layer using BP quite easily – train the first layer, then freeze its weights. Add the second layer, then train it. After each additional layer is trained, it is frozen and the new layer added. This is know as *greedy layer-wise* training. The result is a *stacked* autoencoder.

We'll discuss how stacked autoencoders work to build feature detectors and how they can be used in time-series prediction.

Quiz 7B:

1. * “Feature detectors can't develop in shallow networks” Do you agree with this statement?

2. * Just from what you've learned so far about them, how might autoencoders be used for dimensional reduction?

7.3 Autoencoders

An autoencoder is an ordinary perceptron-based network trained specially to do one task – to *reproduce its input*. Figure 7.1 shows the parts of the autoencoder:

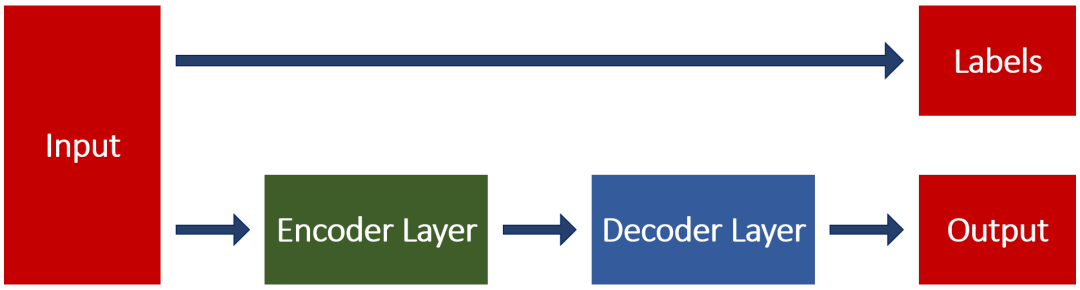

Figure 7.1 – An autoencoder. The Encoder (green) portion is valuable. The Decoder (blue) is discarded after training. The Labels are the Inputs.

An autoencoder has two components:

1. **The Encoder** is an ordinary perceptron (single) layer. This is the valuable part of the autoencoder. Typically, the size of the Encoder is as large as or smaller than the dimension of the input.

2. **The Decoder** is a linear combiner whose dimensions must match that of the input. This layer is trained together with the Encoder, but it is discarded after training. Its only use it in training.

Note that the autoencoder is trained to reproduce its input, since the labels are just the inputs.

7.3 Feature Detectors

During training, the Encoder is forced to develop an internal representation of the input, captured in its weights, and the Decoder then maps this internal representation back into the input.

To put it more concretely, the Encoder transforms the input $\boldsymbol{x}$ into outputs $U_k(\boldsymbol{x})$ while the Decoder performs the inverse transformation:

$$\boldsymbol{x}=D(U_1(\boldsymbol{x})\ldots U_p(\boldsymbol{x}))$$

Since the Decoder layer is just a linear combiner, *D* can be represented by its weights:

$$x_j=\sum_{k=1\ldots p} W_{jk}U_k(\boldsymbol{x}) \quad \text{and} \quad \boldsymbol{x}=(x_{1.}..x_d)$$

The output of the decoder U_k can be interpreted as signaling the presence (or non-presence) of a sub-feature.

We aren't interested in the Decoder's weights because we don't really want to reconstruct the input. Instead, we want to use the internal representation U_k as an alternative input. It is for this reason that the Decoder's weights are discarded; they are only used in training.

This interpretation is especially apparent if the input dimension *d* is smaller than the number of Encoder outputs, *p.* Then, either some of the input is lost or the Encoder has found a more efficient way of encoding the inputs:

- The former is very much under our control since we can decide how closely we want to model the input in our training regime. We can stop training at any time.
- The latter is precisely what feature detectors do – they aid generalization because they detect regularity in the input signal, therefore being able to compress it better. This makes identifying U_k as signaling the presence of a sub-feature plausible.

You might think that the Encoder and Decoder could just "pass through" the input (ie, $U \equiv I$, the trivial identity transformation), but this is not possible. (Quiz 7C.1) In other words, any representation developed by a perceptron-type layer is very likely to be non-trivial.

Quiz 7C:

1. * In Figure 7.1, why can't the Encoder/Decoder layers "pass through" the input forming a trivial identity transformation?

7.4 Adding Layers – Stacked Autoencoders

To add a second Encoding layer, we need to:

1. Discard the old Decoder,
2. Freeze the first Encoder's weights,
3. Add the new Encoder
4. Add a new Decoder
5. Train just the new Encoder and Decoder layers using the output of the old Encoder as labels.

Figure 7.2 illustrates this new configuration:

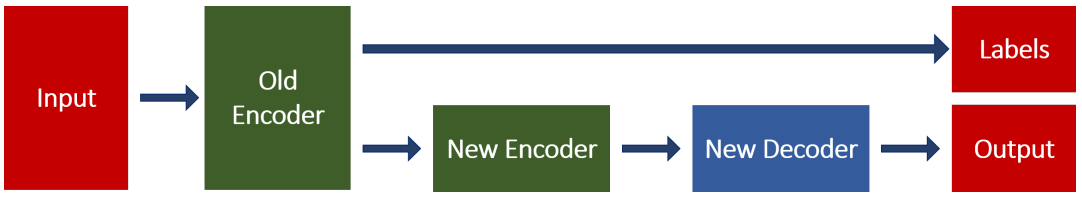

Figure 7.2 – Adding a Layer. Note the old Decoder was discarded and the Old Encoder's weights are frozen. Labels are outputs of the Old Encoder.

Again, just the Encoders (old and new) are valuable. The old Decoder is discarded before beginning the second round of training; the new Decoder is eventually discarded after the new Encoder layer is trained.

7.5 Benefits of Stacked Autoencoders

If we add successive Encoding layers with smaller dimensions, ie,

$$d > p_1 > p_2 \ldots$$

Then we can force the Encoders to compress the inputs more and more. The tradeoff between generalization of the inputs (in terms of feature detectors) and discrepancy between input and transformed input are both completely under our control.

The reduced input dimension serves two purposes:

1. It solves the problem of the curse of high dimensions (refer to Lecture 5)

2. By reducing making the size of each successive transformed input smaller than the original input you can force the formation of successively more compressive feature detectors.

For (2) to work, you may need to reduce the size of each new layer gradually. We recommend a 2/3 rule – reduce the next layer by no more than 2/3. But you should experiment to get the best results.

It isn't always necessary to “force” the formation of feature detectors through compression. Evidence suggests that even if the stack has,

$$d = p_1 = p_2 \ldots$$

good generalization is often still achieved. The lesson is that depth is valuable. Of course, in this case, you would lose the dimensional reduction property which is still useful for prediction.

7.6 Using the Encoder Stack

To use the trained encoder stack, you need to use it in conjunction with a separate predictive network, as in Figure 7.3:

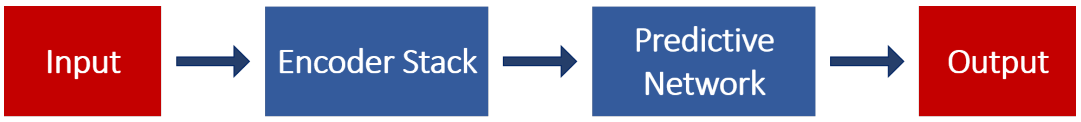

Figure 7.3 – Using an Encoder Stack. The Encoder's weights can either be frozen during training of the Predictive Network, or trained with a much smaller base learning rate, (eg, 1/10 that of the predictive network)

The output of the last Encoder layer is used as input into another predictive network, which needs to be trained separately. This predictive system is usually a shallow 2-layer network or a recurrent network like an LSTM. We will discuss this in Lecture 10.

7.7 Some Advice

How deep should you make your Stacked Autoencoders? Our advice is to make incremental changes to your system and track performance carefully. Stop adding layers when your performance bottoms out. Don't start with very deep networks, instead experiment with a 3 or 5 layer SAE.

Lab 7: Stacked Auto-encoders

In this lab, you will

1. Create a 3-layer autoencoder in prototext and train it with the ae task

Instructions

<u>NetState</u>
Each time a network is loaded from a prototext file by Caffe, the "state" of the network (NetState) can be defined. In the prototext network definitions, each layer can be included or excluded based on the NetState. An example is

```
include: {
        phase: TEST
}
```

which only includes the layer if the current NetState is in the `TEST` phase. The NetState has 3 parts, giving three types of rules to include/exclude the layers

1. **Phase**: This is the example given above. There are only 2 phases – `TRAIN` and `TEST`. Layers are included/excluded only if their phase matches the network's. By default, the phase during training is `TRAIN` and is `TEST` during testing.

2. **Level**: This is an integer value. Layers can be set with `min_level` and/or `max_level` rules to specify the minimum or maximum network level for the layer to be included/excluded.

3. **Stage**: Stages are user-specified strings that are checked for a match between the layer's stage and the network's. Layers are included/excluded based on the `stage` and `not_stage` rules. All the stages specified in `stage` and none of the stages in `not_stage` can match the network's stage for this rule to be activated and the layer included/excluded.

These rules can be used to include or exclude a layer when the rules are met by specifying them in the `include` or `exclude` properties of the layer. For example, to exclude a layer when the NetState level is above 4, you can use

```
include: {  max_level: 4  }
```

or

```
exclude: {  min_level: 5  }
```

Multiple include or exclude rules (but not both!) may be specified for a layer. The layer will be included/excluded if any of the specified rules are satisfied. Example:

```
layer {
     ...
     include: {  max_level: 2  }
     include: {  max_level: 3  }
     include: {  max_level: 4  }
}
```

is equivalent to

```
layer {
     ...
     include: {  max_level: 4  }
}
```

The NetState during training and testing may be set with the `train_state` and `test_state` properties in `solver.prototxt`.

Example:

```
                  train_state { stage: "train" }
                   test_state { stage: "val" }
```

By default, the `train_state` is assigned the phase `TRAIN` and `test_state` has the phase `TEST`.

<u>Autoencoder</u>
The autoencoders are trained separately in its own experiment folder. Once the autoencoders are trained, the models are saved in the `~/caffe/weights/` folder to be used as initial weights for other NNs being trained to do predictions. This lab is focused on the creation and training of the autoencoder. In the next lab, you will start applying autoencoders for generating predictions.

As described in the lecture, autoencoders are trained layer by layer. Multiple training stages are required, starting from a 1 layer network and adding one more layer in the next stage, until the required number of layers is reached. At each training stage, only the newly-added layers are trained, while the weights of the lower layers are ‘frozen’.

The NetState level is used to keep track of the current stage, eg level 1 is for the 1-layer network, level 2 is for the 2-layer network, and so on. All these networks will be defined in the same `train.prototxt` file using the `min_level` and `max_level` rules to include the appropriate layers.

Preprocessing layers
Use the `dengue-2` dataset and the following layers in `train.prototxt`

1. Input: only the data blob is needed (no need for label since the autoencoder uses its input as the label). Set the batch size to the size of the training dataset and combined (train+test) dataset for the train and test phase layers respectively

2. Norm: normalize each variable (axis 1) separately in the data blob. Set the length to the size of the training dataset (714)

3. Slice (and Silence): Slice off the training dataset during the test phase. Use a Power layer to rename the blob during the train phase.

4. Flatten: this layer converts axes after the `axis` param of its input blob to a single axis. The dimensions of axes before the `axis` param are preserved. We use this to flatten the blob to a 1-D vector.

```
layer {
     name: "flat"
     type: "Flatten"
     bottom: "sliced-blob"
     top: "flat-blob"
     flatten_param {
          axis: 1
     }
}
```

Use the same layers as in `train.prototxt` for `test.prototxt`, with these changes

- Delete the train phase input layer
- Set the `source` parameter of the remaining input layer to `val.txt`
- Remove the Slice, Power and Silence layers (from point 3 above)

The output blob `flat-blob` is used as input to the autoencoder layers.

Level 1
We start with the 1-layer autoencoder. Notice the layer inclusion rules for

each layer below. There are two copies of the `Encoder0` layer. The first copy is for NetState level 1 when the layer will train normally, while the second copy is for level 2 and higher and the layer weights are ‘frozen’. This is done by the two `param` sections (the first applies to the weights and the second for the biases) which set the decay multiplier (`decay_mult`, part of regularization: see Lecture 10) and learning rate multiplier (`lr_mult`) to 0.

The decoder and loss layer are only used in this level. The `num_output` of the decoder layer needs to be the size of the input to the `Encoder0` layer (size of `flat-blob` blob = window size (20) * 3 = 60).

```
layer {
	name: "Encoder0"
	type: "InnerProduct"
	bottom: "flat-blob"
	top: "e0"
	include: {
		max_level: 1
		min_level: 1
	}
	inner_product_param {
		axis: 1
		num_output: 36
		bias_filler {
			type: "constant"
			value: 1
		}
		weight_filler {  type: "xavier"  }
	}
}

layer {
	name: "Encoder0"
	type: "InnerProduct"
	bottom: "flat-blob"
	top: "e0"
	include: {  min_level: 2  }
	inner_product_param {
		axis: 1
		num_output: 36
		bias_filler {
			type: "constant"
			value: 0
		}
		weight_filler {  type: "xavier"  }
	}
	param {
		decay_mult: 0
```

```
                lr_mult: 0
        }
        param {
                decay_mult: 0
                lr_mult: 0
        }
}

layer {
        name: "Encoder1"
        bottom: "e0"
        include: {  min_level: 1  }
        top: "t0"
        type: "TanH"
}

layer {
        name: "Decoder0"
        type: "InnerProduct"
        bottom: "t0"
        top: "d0"
        include: {
                max_level: 1
                min_level: 1
        }
        inner_product_param {
                axis: 1
                bias_filler {
                        type: "constant"
                        value: 0
                }
                num_output: 60
                weight_filler {  type: "xavier"  }
        }
}

layer {
        name: "Loss0"
        type: "EuclideanLoss"
        bottom: "d0"
        bottom: "flat-blob"
        top: "loss"
        include: {
                max_level: 1
                min_level: 1
        }
}
```

Level 2

The next level has the same additional layer structure as level 1. This time the input to the encoder layer and the loss target are the output of `Encoder1` layer (blob `t0`). The `num_output` of the decoder layer also has to be equal to the size of the `t0` blob.

```
layer {
        name: "Encoder2"
        type: "InnerProduct"
        bottom: "t0"
        top: "e1"
        include: {
                max_level: 2
                min_level: 2
        }
        inner_product_param {
                axis: 1
                num_output: 20
                bias_filler {
                        type: "constant"
                        value: 0
                }
                weight_filler {  type: "xavier"  }
        }
}

layer {
        name: "Encoder2"
        type: "InnerProduct"
        bottom: "t0"
        top: "e1"
        include: {  min_level: 3  }
        inner_product_param {
                axis: 1
                num_output: 20
                bias_filler {
                        type: "constant"
                        value: 0
                }
                weight_filler {  type: "xavier"  }
        }
        param {
                decay_mult: 0
                lr_mult: 0
        }
        param {
                decay_mult: 0
                lr_mult: 0
        }
}

layer {
        name: "Encoder3"
        bottom: "e1"
        include: {  min_level: 2  }
        top: "t1"
        type: "TanH"
}

layer {
        name: "Decoder1"
        type: "InnerProduct"
        bottom: "t1"
        top: "d1"
        include: {
                max_level: 2
                min_level: 2
        }
        inner_product_param {
                axis: 1
                bias_filler {
                        type: "constant"
                        value: 0
                }
                num_output: 36
                weight_filler {  type: "xavier"  }
        }
}

layer {
        name: "Loss1"
        type: "EuclideanLoss"
        bottom: "d1"
        bottom: "t0"
        top: "loss"
        include: {
                max_level: 2
                min_level: 2
        }
}
```

Level 3
Refer to the previous 2 levels and try to make this level. Be very careful when making the changes as any mistakes will have to be debugged later on. Change the `num_output` of the inner product layers from all 3 stages to try out different layer size combinations.

Test network
For `test.prototxt`, just copy the network from `train.prototxt` and delete the final loss layer. Rename the output blob of the decoder layer as “prediction”.

Add a Power layer to rename the `flat-blob` blob to `label`. Without any parameters set, the power layer does not modify the contents of the blob.

```
layer {
        name: "no-op"
        type: "Power"
        bottom: "t1"
        top: "label"
}
```

The `test.prototxt` network will be run by the `ac test` command with NetState level 3 (as given in `config.txt` below). Hence level include rules are not needed for the above Power layer.

`config.txt`
Add this line to tell autocaffe how many training stages there are:

```
LEVELS: 3
```

Set the test phase and prediction run iterations to 1.
In `config.txt`:

```
TEST_ITERS: 1
```

In `solver.prototxt`:

```
test_iter: 1
```

`report.tex_template` & `metrics`
These files are provided in `~/caffe/datathon-files/lab7/`, please copy them into your experiment folder (`~/caffe/experiments/lab7`).

The report template includes the plot of training losses of each training stage. The `$LOSS_AE` string in the template will be substituted with the loss of the final level (ie reconstruction of the input to the third layer encoder) before the pdf report is generated.

The `metrics` file provided had irrelevant functions to autoencoder training removed, like train and test loss.

Training using `ae` task

Step 1: do a `prep`

```
ac prep lab7
```

Step 2: add `ae` task

```
ac add-task lab7/* ae
```

Step 3: check on progress by using

```
ac tasks
```

A typical entry has 3 parts and looks like this:

```
7/1 ae 2
```

and has this format: `<experiment directory> <task> <current level>`
The third part (current level) will not be present if the task is newly added.

The `ae` task does the following for you

1. Creates subfolders in the experiment's results folder (eg `~/caffe/results/lab7/1`) – one for each training level
2. Sets up each subfolder for training at the appropriate level by copying the needed files and adding a `LEVEL` entry to `config.txt`. For level 2 onwards, it searches for the trained model in the previous level and writes its path in a `WEIGHTS` entry in `config.txt`
3. Adds the job for the current to the queue, waits for it to complete, then starts the job for the next level
4. After the final level is complete, it copies the final trained model to the main folder (eg `~/caffe/results/lab7/1`) and runs the `test` and `post` commands.

After the task is done, generate the pdf report using

```
ac report lab7/*
```

Log entries are written to a `tasklog` file in the experiment folder – be sure to take a look at the log file when troubleshooting your experiments.

Note: When a task disappears from the `ac task` list, it could mean that the task has completed, but it may also have errored out or may still be running its post-processing/plotting functions. Check the `tasklog` file for error messages or a task complete line to distinguished between the two. If there are no error messages but the task has not completed, the task is probably still running, so give it some time to complete before checking the `tasklog` again. If still nothing appears, check the `train.log` and `test.log` files of each stage (contained in their own subfolders).